## **Question 1**: What are React hooks? How do useState() and useEffect() hooks work in functional components?

Ans - React hooks are special functions that allow developers to use state and other React features (like side effects, context) in **functional components**, which were previously limited to only class components. Hooks provide a more concise and reusable way to manage state, lifecycle methods, and other features, making functional components more powerful.

### Two Important React Hooks: `useState()` and `useEffect()`

#### 1. `useState()` Hook

The `useState()` hook is used to add state to functional components. In class components, state is managed using the `this.state` object and updated using `this.setState()`. The `useState()` hook simplifies this in functional components.

#### 2. `useEffect()` Hook

The `useEffect()` hook allows you to perform **side effects** in functional components. Side effects include operations like fetching data, manipulating the DOM, setting up subscriptions, timers, etc. Before hooks, you had to use lifecycle methods in class components,

## **Question 2**: What problems did hooks solve in React development? Why are hooks considered an important addition to React?

Ans - React hooks addressed several significant challenges and limitations that developers faced when using **class components** in React. Before the introduction of hooks in React 16.8, developers were constrained by a number of issues related to state management, component reusability, lifecycle methods, and code complexity. Here are some of the key problems that hooks solved:

React hooks are considered a major evolution in React development because they simplify component structure, improve reusability, and make code more predictable and manageable. Hooks allow developers to manage state and side effects in functional components, addressing many of the limitations of class components. By enabling better code organization, modularity, and performance, hooks have become an essential tool for building modern React applications.

## Why Are Hooks Considered an Important Addition to React?

1. **Simplify Code and Make It More Readable**: Hooks make it easier to write simpler, more readable React components. Functional components are typically easier to reason about, and hooks reduce the boilerplate code that class components required. This results in smaller, more maintainable codebases.
2. **Improve Code Reusability and Composition**: By using hooks like `useState`, `useEffect`, and custom hooks, developers can create reusable, composable logic across different components. Hooks allow code to be more modular, enabling better separation of concerns.
3. **Enhanced Performance**: React hooks, especially `useState` and `useEffect`, give developers more fine-grained control over how components re-render. For example, hooks allow you to prevent unnecessary re-renders by selectively managing the state and side effects. This can lead to performance improvements in large applications.
4. **Ease of Understanding for New Developers**: Functional components with hooks have a more intuitive structure for new developers compared to class components. With less boilerplate and clearer syntax, hooks make it easier for developers to get started with React and write clean, modern code.
5. **Better Testing and Debugging**: Since hooks are just functions, they can be independently tested and reused. This improves testability and helps reduce bugs, especially in larger applications where state and side effects are complex. React DevTools also provide enhanced support for inspecting hooks.
6. **Consolidation of Lifecycle Methods**: The `useEffect` hook combines several lifecycle methods into one, making it easier to understand and manage component lifecycles. This reduces the cognitive load of working with multiple lifecycle methods and provides a more declarative way of managing side effects

## **Question 3**: What is useReducer ? How we use in react app?

Ans - `useReducer` is a React hook that is often used as an alternative to `useState` for managing more complex state logic in functional components. While `useState` is great for managing simple states (like a number or a string), `useReducer` is more suited for scenarios where the state is more complex, involves multiple sub-values, or requires state transitions based on actions (like in state management frameworks such as Redux).

`useReducer` allows you to manage state updates through **dispatching actions**, which are handled by a **reducer function**. This provides a more structured way to handle state changes, making it easier to scale when dealing with complex state logic.

### Syntax of `useReducer`

The basic syntax for using `useReducer` is:

```
javascript
Copy code
const [state, dispatch] = useReducer(reducer, initialState);
```

- **`state`**: The current state value, which can be any data type (object, array, etc.).
- **`dispatch`**: A function that allows you to dispatch actions to update the state.
- **`reducer`**: A function that describes how the state should change based on the dispatched action.
- **`initialState`**: The initial state value.

### `useReducer` Components:

1. **Reducer Function**: A pure function that takes the current state and an action, and returns a new state.
    - Signature: `(state, action) => newState`
2. **Dispatch Function**: A function that triggers the reducer and sends actions to it to update the state.
    - You can think of it as a way to **notify the reducer** that something has happened (an action).

### How `useReducer` Works:

1. The `useReducer` hook takes a **reducer function** and an **initial state**.
2. When an action is dispatched, the reducer function receives the current state and the action and returns a **new state**.
3. The component is re-rendered when the state changes, just like with `useState`.

## **Question 6 :** What is useRef ? How to work in react app?

Ans - The `useRef` hook is a React hook that provides a way to access and interact with **DOM elements** or **persist values** across renders in a **functional component** without triggering a re-render.

It returns an object with a `current` property, which can hold a reference to a DOM element or any mutable value that you want to persist across renders.

## What is `useRef`?

The `useRef` hook is a React hook that provides a way to access and interact with **DOM elements** or **persist values** across renders in a **functional component** without triggering a re-render.

It returns an object with a `current` property, which can hold a reference to a DOM element or any mutable value that you want to persist across renders.

## Key Use Cases of `useRef`:

1. **Accessing DOM Elements**: You can use `useRef` to get a reference to a DOM element and interact with it directly, like focusing on an input field or measuring the size of an element.
2. **Persisting Mutable Values**: You can store values that need to persist between renders, but don't require a re-render when updated. This is useful for keeping track of timers, previous props/state, or any other values that don't affect the component's rendering.

## Syntax of `useRef`:

```
javascript
Copy code
const myRef = useRef(initialValue);
```

- `myRef` is the reference object returned by `useRef`.
- `initialValue` is an optional argument that sets the initial value of the reference object.

## How Does `useRef` Work?

1. **Persistent Mutable Values**: When you assign a value to `myRef.current`, it persists across renders but does not cause a re-render when the value changes.
2. **Accessing DOM Elements**: If you attach the `ref` object to a DOM element, `myRef.current` will point to that DOM element. You can then manipulate or read properties of the DOM element directly.

## Task 1:

## • Create a functional component with a counter using the useState() hook. Include buttons to increment and decrement the counter.

## Ans -

```
import React, { useState } from 'react';
```

```
const Counter = () => {
  const [count, setCount] = useState(0);

  const increment = () => setCount(count + 1);

  const decrement = () => setCount(count - 1);
  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
};
export default Counter;
```

## Task 2:

## • Use the useEffect() hook to fetch and display data from an API when the component mounts.

Ans - 

```
import React, { useState, useEffect } from 'react';
const FetchDataComponent = () => {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
```

```
useEffect(() => {
  const fetchData = async () => {
    try {
      const response = await fetch('https://jsonplaceholder.typicode.com/posts');
      if (!response.ok) {
        throw new Error('Failed to fetch data');
      }
      const data = await response.json();
      setData(data);  // Set the fetched data into state
    } catch (error) {
      setError(error.message);  // Set error if fetch fails
    } finally {
      setLoading(false);  // Set loading to false after fetching data
    }
  };

  fetchData(); // Call the fetchData function when the component mounts
}, []);  // Empty dependency array to run the effect only once when the component mounts

if (loading) {
  return <p>Loading...</p>;  // Show loading text while data is being fetched
}

if (error) {
  return <p>Error: {error}</p>;  // Display an error message if an error occurs
}
```

```
  return (
    <div>
      <h1>Posts</h1>
      <ul>
        {data.map(post => (
          <li key={post.id}>{post.title}</li>  // Displaying the title of each post
        ))}
      </ul>
    </div>
  );
};

export default FetchDataComponent;
```

## Task 3:

## • Create react app with use of useSelector & useDispatch.

## Ans -

```
import { createSlice } from '@reduxjs/toolkit';

export const counterSlice = createSlice({
  name: 'counter',
  initialState: { value: 0 },
  reducers: {
    increment: (state) => {
      state.value += 1;
    },
    decrement: (state) => {
```

```
      state.value -= 1;
    },
    incrementByAmount: (state, action) => {
      state.value += action.payload;
    }
  }
});
export const { increment, decrement, incrementByAmount } = counterSlice.actions;
export default counterSlice.reducer;



// src/redux/store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

// Create and export the store
export const store = configureStore({
  reducer: {
    counter: counterReducer
  }
});

// src/redux/store.js
import { configureStore } from '@reduxjs/toolkit';
import counterReducer from './counterSlice';

// Create and export the store
```

```
export const store = configureStore({
  reducer: {
    counter: counterReducer
  }
});



// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import './index.css';
import App from './App';
import { store } from './redux/store';

ReactDOM.render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById('root')
);

// src/App.js
import React from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { increment, decrement, incrementByAmount } from './redux/counterSlice';
```

```
function App() {
  const dispatch = useDispatch();
  const count = useSelector((state) => state.counter.value);

  return (
    <div className="App">
      <h1>Counter: {count}</h1>
      <button onClick={() => dispatch(increment())}>Increment</button>
      <button onClick={() => dispatch(decrement())}>Decrement</button>
      <button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>
    </div>
  );
}

export default App;
```

## Task 4:

• Create react app to avoid re-renders in react application by useRef ?

```
import React, { useState, useRef } from 'react';

const App = () => {
  // State that will trigger re-renders
  const [countState, setCountState] = useState(0);

  // Ref that will not trigger re-renders
```

```
const countRef = useRef(0);

// Function to increment the state value
const incrementState = () => {
  setCountState(countState + 1);
};

// Function to increment the ref value
const incrementRef = () => {
  countRef.current += 1;
  console.log("Ref value:", countRef.current);  // Log the updated ref value
};

return (
  <div className="App">
    <h1>useState vs useRef</h1>
    <div>
      <h2>State Count (useState)</h2>
      <p>Count using useState: {countState}</p>
      <button onClick={incrementState}>Increment State</button>
    </div>
    <div>
      <h2>Ref Count (useRef)</h2>
      <p>Count using useRef: {countRef.current}</p>
      <button onClick={incrementRef}>Increment Ref</button>
    </div>
  </div>
```

```
  );
};

export default App;
```

**Question 1**: What is React Router? How does it handle routing in single-page applications?
**Ans –**

**React Router** is a standard library used for routing in React applications. It allows you to implement dynamic routing, which helps navigate between different views or components within a single-page application (SPA). React Router enables you to map URLs to specific components, and it efficiently handles the navigation without requiring full page reloads, which is a key feature of single-page applications.

How its handle -

In single-page applications, the entire application is loaded in a single HTML page, and content changes dynamically based on user interactions (like clicking links). React Router provides a way to change the view or component shown based on the URL, while keeping the page from reloading.

## **Question 2**: Explain the difference between BrowserRouter, Route, Link, and Switch components in React Router.

Ans –

differences between `BrowserRouter`, `Route`, `Link`, and `Switch` components in React Router:

1. **`BrowserRouter`**:
   - Wraps the application and enables routing using the HTML5 history API.
   - Needed to manage URL and navigation.
2. **`Route`**:
   - Maps a URL path to a specific component.
   - Renders the associated component when the URL matches the path.
3. **`Link`**:
   - Provides navigation between different routes without page reloads.
   - Similar to an anchor tag (`<a>`), but updates the URL and renders components.
4. **`Switch`**:
   - Groups multiple `Route` components and renders only the first matching route.

- Ensures that only one route is rendered at a time.

## Task 1:

## • Set up a basic React Router with two routes: one for a Home page and one for an About page. Display the appropriate content based on the URL.

## Ans-

```
import React from 'react';

import { BrowserRouter as Router, Route, Link, Switch } from 'react-router-dom';


// Components for the Home and About pages

const Home = () => <h2>Home Page</h2>;

const About = () => <h2>About Page</h2>;


const App = () => {

  return (

    <Router>

      <div>

        <nav>

          <ul>

            <li><Link to="/">Home</Link></li>
```

```
          <li><Link to="/about">About</Link></li>
        </ul>
      </nav>        <Switch>
        <Route exact path="/" component={Home} />
        <Route path="/about" component={About} />
      </Switch></div>  </Router>
  );
};
export default App;
```

## Task 2:

## • Create a navigation bar using React Router's Link component that allows users to switch between the Home, About, and Contact pages.

Ans –

```
import React from 'react';
import { BrowserRouter as Router, Route, Link, Switch } from 'react-router-dom';
// Components for Home, About, and Contact pages
const Home = () => <h2>Home Page</h2>;
const About = () => <h2>About Page</h2>;
const Contact = () => <h2>Contact Page</h2>;
const App = () => {
  return (
    <Router>
```

```
      <div>

        <nav>
          <ul>
            <li><Link to="/">Home</Link></li>
            <li><Link to="/about">About</Link></li>
            <li><Link to="/contact">Contact</Link></li>
          </ul>
        </nav>

        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
        </Switch>
      </div>
    </Router>
  );
};
export default App;
```

///

```
import React, { useEffect, useState } from 'react';

const PostList = () => {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    fetch('http://localhost:5000/posts')
      .then(response => response.json())
      .then(data => setPosts(data));
  }, []);

  return (
    <ul>
      {posts.map(post => (
        <li key={post.id}>{post.title}</li>
      ))}
    </ul>
  );
};
```

## **Question 3**: How do you fetch data from a Json-server API in React? Explain the role of fetch() or axios() in making API requests.

Ans -

- **`fetch()`**: It is a native JavaScript function for making HTTP requests. It returns a promise and requires manual handling of response parsing and errors.

```
fetch('http://localhost:3000/posts')
  .then(response => response.json())
  .then(data => setData(data));
```

- **`axios()`**: It is a more feature-rich library that simplifies requests, automatically parses JSON, and handles errors more gracefully.

```
axios.get('http://localhost:3000/posts')

  .then(response => setData(response.data));
```

## **Question 4:** What is Firebase? What features does Firebase offer?

Ans - **Firebase** is a platform developed by Google for building and managing web and mobile applications. It provides a suite of backend services to simplify app development, including real-time databases, authentication, hosting, and cloud storage.

### Key Features of Firebase:

1. **Firebase Realtime Database**: A NoSQL cloud database for storing and syncing data in real-time across all clients.
2. **Firebase Firestore**: A flexible, scalable NoSQL cloud database for storing, syncing, and querying app data.
3. **Firebase Authentication**: Provides easy-to-use SDKs for user authentication via email, social media, or phone numbers.
4. **Firebase Hosting**: A service to deploy and host static websites and web apps with SSL and global CDN.
5. **Firebase Cloud Functions**: Allows serverless backend code to run in response to events triggered by Firebase features and HTTP requests.
6. **Firebase Cloud Storage**: A solution for storing user-generated content like photos and videos.
7. **Firebase Analytics**: Provides insights into user behavior and app performance.

## **Question 5**: Discuss the importance of handling errors and loading states when working with APIs in React

Ans - Handling errors and loading states when working with APIs in React is crucial for ensuring a smooth user experience. **Loading states** inform users that data is being fetched,

preventing confusion and frustration. **Error handling** ensures that users are notified if something goes wrong (e.g., network issues or server errors), preventing app crashes and improving reliability. Proper handling of both ensures that the app remains responsive, provides clear feedback, and can recover gracefully from failures.

## Task 1:

- Create a React component that fetches data from a public API (e.g., a list of users) and displays it in a table format.
- Create a React app with Json-server and use Get , Post , Put , Delete & patch method on Json-server API.

Ans - import React, { useState, useEffect } from 'react';

```
const UserTable = () => {
  // State to hold the user data and loading state
  const [users, setUsers] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);
  // Fetch data from the API when the component mounts
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://jsonplaceholder.typicode.com/users');
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
```

```
      const data = await response.json();
      setUsers(data); // Set the fetched users to state
    } catch (error) {
      setError(error.message); // Set error if there's an issue with fetching
    } finally {
      setLoading(false); // Set loading to false when data is fetched or error occurs
    }
  };

  fetchData();
}, []); // Empty dependency array to fetch data only once when the component mounts

// Render loading, error, or the user table
if (loading) return <div>Loading...</div>;
if (error) return <div>Error: {error}</div>;

return (
  <div>
    <h1>User List</h1>
    <table border="1" cellPadding="5">
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Email</th>
```

```
            <th>Phone</th>
            <th>Website</th>
          </tr>
        </thead>
        <tbody>
          {users.map(user => (
            <tr key={user.id}>
              <td>{user.id}</td>
              <td>{user.name}</td>
              <td>{user.email}</td>
              <td>{user.phone}</td>
              <td>{user.website}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
};

export default UserTable;

// import React, { useState, useEffect } from 'react';
import axios from 'axios';
```

```
const App = () => {
  const [users, setUsers] = useState([]);
  const [newUser, setNewUser] = useState({ name: '', email: '' });
  const [updatedUser, setUpdatedUser] = useState({ id: '', name: '', email: '' });

  // Fetch users (GET)
  const fetchUsers = async () => {
    try {
      const response = await axios.get('http://localhost:5000/users');
      setUsers(response.data);
    } catch (error) {
      console.error('Error fetching users:', error);
    }
  };

  // Add a new user (POST)
  const addUser = async () => {
    try {
      const response = await axios.post('http://localhost:5000/users', newUser);
      setUsers([...users, response.data]);
      setNewUser({ name: '', email: '' });
    } catch (error) {
      console.error('Error adding user:', error);
    }
  };
```

```
  // Update an existing user (PUT)
  const updateUser = async () => {
    try {
      const response = await axios.put(`http://localhost:5000/users/${updatedUser.id}`,
updatedUser);
      setUsers(users.map(user => (user.id === updatedUser.id ? response.data : user)));
      setUpdatedUser({ id: '', name: '', email: '' });
    } catch (error) {
      console.error('Error updating user:', error);
    }
  };

  // Delete a user (DELETE)
  const deleteUser = async (id) => {
    try {
      await axios.delete(`http://localhost:5000/users/${id}`);
      setUsers(users.filter(user => user.id !== id));
    } catch (error) {
      console.error('Error deleting user:', error);
    }
  };

  // Update user info for PATCH request
  const patchUser = async (id) => {
    try {
```

```
    const patchData = { name: 'Updated Name' }; // Example of partial update
    const response = await axios.patch(`http://localhost:5000/users/${id}`, patchData);
    setUsers(users.map(user => (user.id === id ? response.data : user)));
  } catch (error) {
    console.error('Error patching user:', error);
  }
};

useEffect(() => {
  fetchUsers(); // Fetch users when the component mounts
}, []);

return (
  <div>
    <h1>Users</h1>

    {/* Display User List */}
    <table border="1" cellPadding="5">
      <thead>
        <tr>
          <th>ID</th>
          <th>Name</th>
          <th>Email</th>
          <th>Actions</th>
        </tr>
```

```
  </thead>
  <tbody>
    {users.map((user) => (
      <tr key={user.id}>
        <td>{user.id}</td>
        <td>{user.name}</td>
        <td>{user.email}</td>
        <td>
          <button onClick={() => patchUser(user.id)}>Patch</button>
          <button onClick={() => deleteUser(user.id)}>Delete</button>
        </td>
      </tr>
    ))}
  </tbody>
</table>

{/* Add a new user */}
<div>
  <h2>Add User</h2>
  <input
    type="text"
    placeholder="Name"
    value={newUser.name}
    onChange={(e) => setNewUser({ ...newUser, name: e.target.value })}
  />
```

```
  <input
    type="email"
    placeholder="Email"
    value={newUser.email}
    onChange={(e) => setNewUser({ ...newUser, email: e.target.value })}
  />
  <button onClick={addUser}>Add User</button>
</div>

{/* Update user info */}
<div>
  <h2>Update User</h2>
  <input
    type="text"
    placeholder="User ID"
    value={updatedUser.id}
    onChange={(e) => setUpdatedUser({ ...updatedUser, id: e.target.value })}
  />
  <input
    type="text"
    placeholder="Name"
    value={updatedUser.name}
    onChange={(e) => setUpdatedUser({ ...updatedUser, name: e.target.value })}
  />
  <input
```

```
          type="email"
          placeholder="Email"
          value={updatedUser.email}
          onChange={(e) => setUpdatedUser({ ...updatedUser, email: e.target.value })}
        />
        <button onClick={updateUser}>Update User</button>
      </div>
    </div>
  );
};\
export default App;
```

## Task 2:

- Create a React app crud and Authentication with firebase API.
- Implement google Authentication with firebase API.

Ans -

```
import { initializeApp } from 'firebase/app';
import { getAuth, GoogleAuthProvider } from 'firebase/auth';
import { getFirestore } from 'firebase/firestore';
const firebaseConfig = {
  apiKey: 'YOUR_API_KEY',
  authDomain: 'YOUR_AUTH_DOMAIN',
  projectId: 'YOUR_PROJECT_ID',
```

```
  storageBucket: 'YOUR_STORAGE_BUCKET',
  messagingSenderId: 'YOUR_MESSAGING_SENDER_ID',
  appId: 'YOUR_APP_ID',
};
const app = initializeApp(firebaseConfig);
const auth = getAuth(app);
const firestore = getFirestore(app);
export { auth, firestore, GoogleAuthProvider };
import React, { useState, useEffect } from 'react';
import { onAuthStateChanged } from 'firebase/auth';
import { auth } from './firebaseConfig';
import GoogleAuth from './components/GoogleAuth';

function App() {
  const [user, setUser] = useState(null);
  useEffect(() => {
    const unsubscribe = onAuthStateChanged(auth, (user) => {
      setUser(user);
    });
    return unsubscribe;
  }, []);
  return (
    <div className="App">
      {user ? (
        <div>
```

```
          <h1>Welcome, {user.displayName}</h1>
          <button onClick={() => auth.signOut()}>Sign Out</button>
        </div>
      ) : (
        <GoogleAuth />
      )}
    </div>
  );
}
export default App;
import { collection, addDoc, getDocs, updateDoc, deleteDoc, doc } from 'firebase/firestore';
import { firestore } from './firebaseConfig';
const tasksCollectionRef = collection(firestore, 'tasks')
export const createTask = async (task) => {
  try {
    await addDoc(tasksCollectionRef, task);
  } catch (error) {
    console.error('Error adding document: ', error);
  }
};
export const getTasks = async () => {
  const snapshot = await getDocs(tasksCollectionRef);
  const tasksList = snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));
  return tasksList;
};
```

```
// Update a task
export const updateTask = async (taskId, updatedTask) => {
  const taskDoc = doc(firestore, 'tasks', taskId);
  await updateDoc(taskDoc, updatedTask);
};

// Delete a task
export const deleteTask = async (taskId) => {
  const taskDoc = doc(firestore, 'tasks', taskId);
  await deleteDoc(taskDoc);
};
```

## Task 3:

## • Implement error handling and loading states for the API call. Display a loading spinner while the data is being fetched.

Ans –

```
import React, { useState, useEffect } from 'react';
import { getTasks, createTask, updateTask, deleteTask } from '../firestoreService';
import LoadingSpinner from './LoadingSpinner';
const TaskManager = () => {
  const [tasks, setTasks] = useState([]);
  const [newTask, setNewTask] = useState('');
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState('');
```

```
useEffect(() => {
  const fetchTasks = async () => {
    setLoading(true);
    try {
      const tasksList = await getTasks();
      setTasks(tasksList);
    } catch (error) {
      setError('Failed to load tasks. Please try again later.');
    } finally {
      setLoading(false);
    }
  };
  fetchTasks();
}, []);


const handleAddTask = async () => {
  if (newTask) {
    setLoading(true);
    setError('');
    try {
      await createTask({ name: newTask });
      setNewTask('');
      const updatedTasks = await getTasks();
```

```
      setTasks(updatedTasks);
    } catch (error) {
      setError('Failed to add task. Please try again later.');
    } finally {
      setLoading(false);
    }
  }
};

const handleUpdateTask = async (id, updatedName) => {
  setLoading(true);
  setError('');
  try {
    await updateTask(id, { name: updatedName });
    const updatedTasks = await getTasks();
    setTasks(updatedTasks);
  } catch (error) {
    setError('Failed to update task. Please try again later.');
  } finally {
    setLoading(false);
  }
};

const handleDeleteTask = async (id) => {
  setLoading(true);
```

```
  setError('');
  try {
    await deleteTask(id);
    const updatedTasks = await getTasks();
    setTasks(updatedTasks);
  } catch (error) {
    setError('Failed to delete task. Please try again later.');
  } finally {
    setLoading(false);
  }
};

return (
  <div>
    <h2>Task Manager</h2>
```